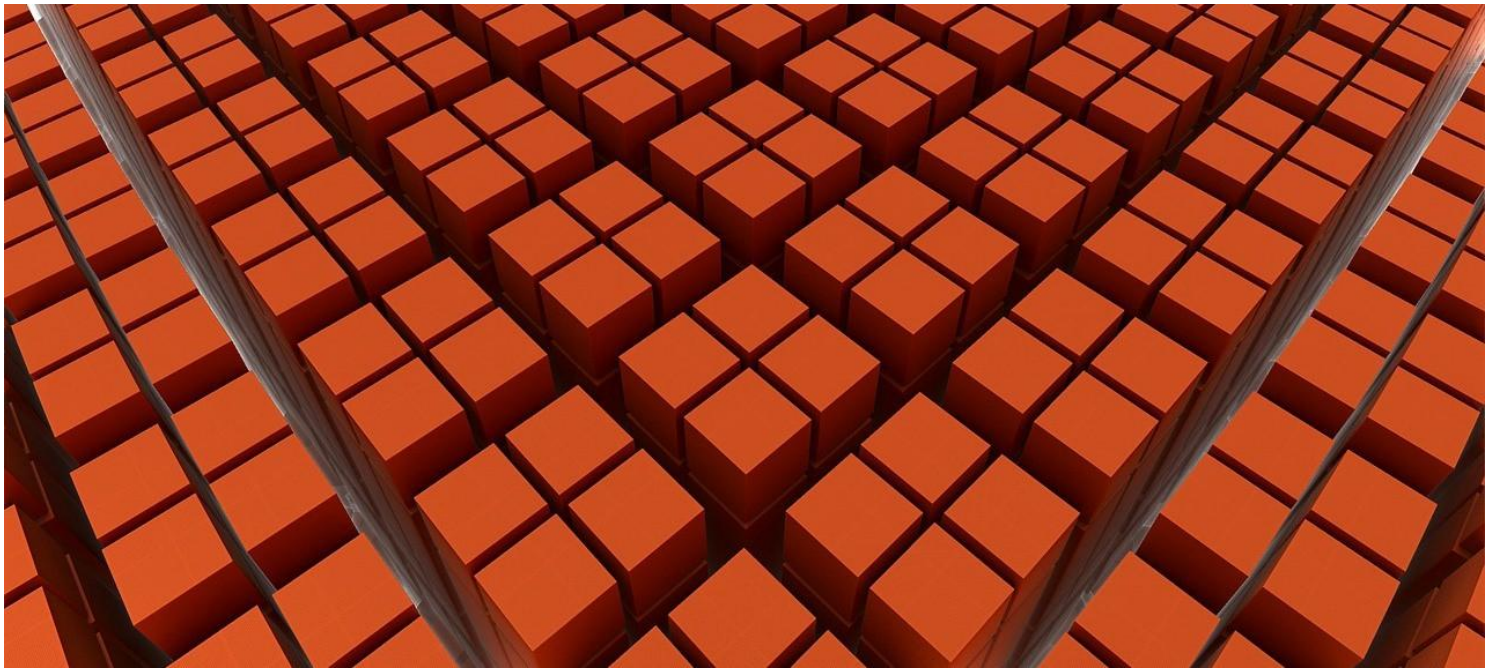
# Language Processing — Text Classification with Universal Embeddings

A Guide to Demystifying Universal Sentence Encoders

Dipanjan (DJ) Sarkar Follow

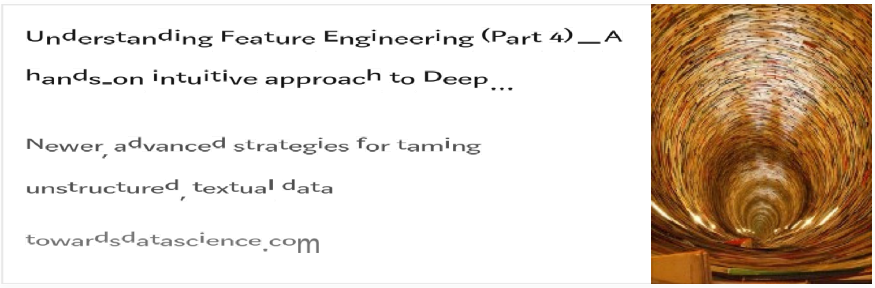
Dec 4, 2018 · 18 min read



## Introduction

Transfer learning is an exciting concept where we try to leverage prior knowledge from one domain and task into a different domain and task. The inspiration comes from us—humans, ourselves—where in, we have an inherent ability to not learn everything from scratch. We transfer and leverage our knowledge from what we have learnt in the past for tackling a wide variety of tasks. With computer vision, we have excellent *big* datasets available to us, like Imagenet, on which, we get a suite of world-class, state-of-the-art pre-trained model to leverage transfer learning. But what about Natural Language Processing?

Therein lies the challenge, considering text data is so diverse, noisy and unstructured. We've had some recent successes with word embeddings including methods like Word2Vec, GloVe and FastText, all of which I have covered in my article ***'Feature Engineering for Text Data'***.




In this article, we will be showcasing several state-of-the-art generic sentence embedding encoders, which tend to give surprisingly good performance, especially on small amounts of data for transfer learning tasks as compared to word embedding models. We will be covering the following models:

- **Baseline Averaged Sentence Embeddings**
- **Doc2Vec**
- **Neural-Net Language Models (Hands-on Demo!)**
- **Skip-Thought Vectors**
- **Quick-Thought Vectors**
- **InferSent**
- **Universal Sentence Encoder**

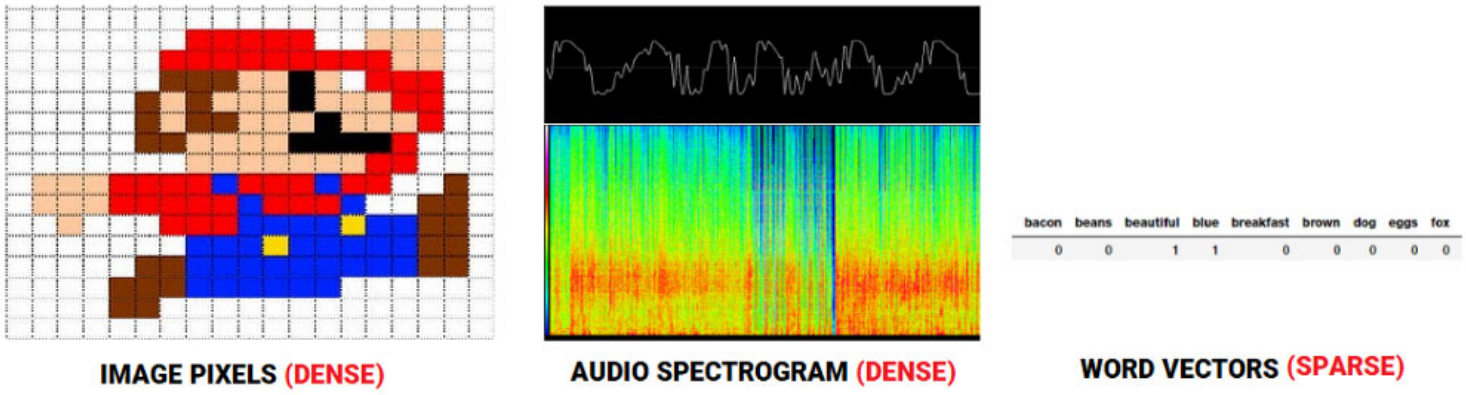
We will try to cover essential concepts and also showcase some hands-on examples leveraging Python and Tensorflow, in a text classification problem focused on sentiment analysis!

## Why are we crazy for Embeddings?

What is this sudden craze behind embeddings? I'm sure many of you might be hearing it everywhere. Let's clear up the basics first and cut through the hype.

> *An embedding is a fixed-length vector typically used to encode and represent an entity (document, sentence, word, graph!)*

I've talked about the need for embeddings in the context of text data and NLP in *one of my previous articles*. But I will briefly reiterate this here for the sake of convenience. With regard to speech or image recognition systems, we already get information in the form of rich dense feature vectors embedded in high-dimensional datasets like audio spectrograms and image pixel intensities. However, when it comes to raw text data, especially count-based models like Bag of Words, we are dealing with individual words, which may have their own identifiers, and do not capture the semantic relationship among words. This leads to huge sparse word vectors for textual data and thus, if we do not have enough data, we may end up getting poor models or even overfitting the data due to the curse of dimensionality.

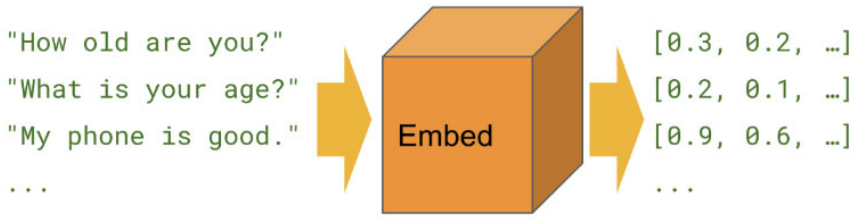



Comparing feature representations for audio, image and text

***Predictive methods*** like ***Neural Network based language models*** try to predict words from its neighboring words looking at word sequences in the corpus and in the process, it learns distributed representations, giving us dense word embeddings.

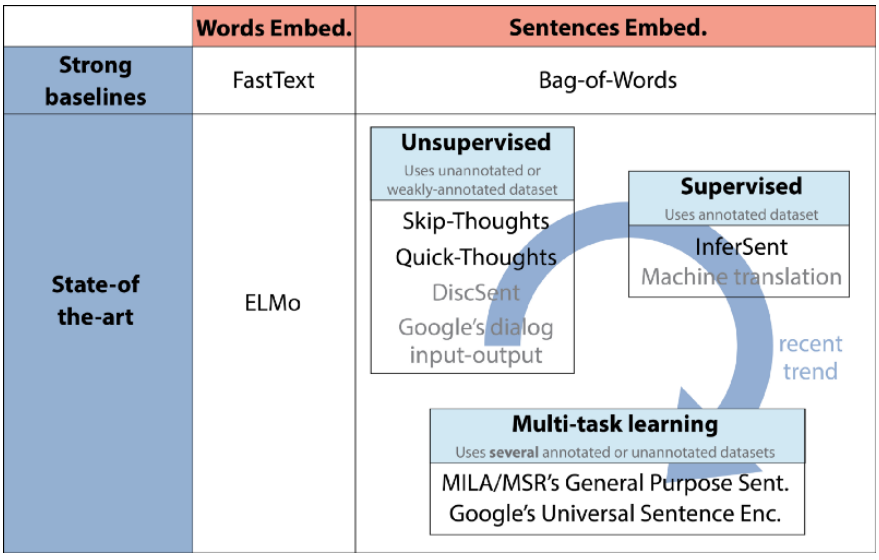
Now you might be thinking, big deal, we get a bunch of vectors from text. What now? Well, this craze for embeddings is that, if we have a good numeric representation of text data which captures even the context and semantics, we can use it for a wide variety of downstream real-world tasks like sentiment analysis, text classification, clustering, summarization, translation and so on. The fact of the matter is,

machine learning or deep learning models run on numbers, and embeddings are the key to encoding text data that will be used by these models.



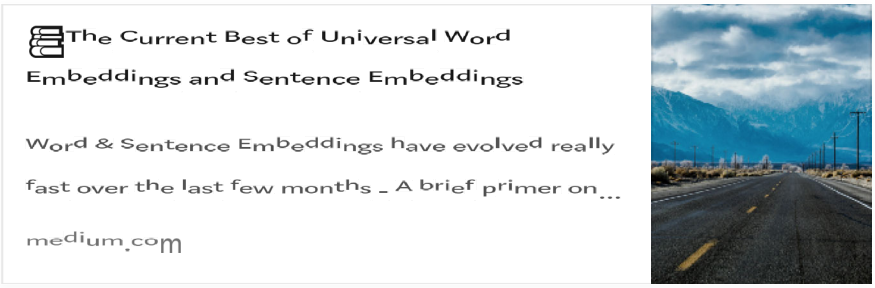

Text Embeddings

A big trend here has been finding out so-called ***'Universal Embeddings'*** which are basically pre-trained embeddings obtained from training deep learning models on a huge corpus. This enables us to use these pre-trained (generic) embeddings in a wide variety of tasks including, scenarios with constraints like lack of adequate data. This is a perfect example of transfer learning, leveraging prior knowledge from pre-trained embeddings to solve a completely new task! The following figure showcases some recent trends in Universal Word & Sentence Embeddings, thanks to *an amazing article* from the folks over at *HuggingFace*!




Recent Trends in Universal Word & Sentence Embeddings (Source: https://medium.com/huggingface/universal_word_sentence_embeddings_ce48ddc8fc3a)

Definitely, some interesting trends in the above figure including, Google's Universal Sentence Encoder, which we will be exploring in detail in this article! I definitely recommend readers to check out *the article on universal embedding trends* from *HuggingFace*.




Now, let's take a brief look at trends and developments in word and sentence embedding models before diving deeper into Universal Sentence Encoder.

## Trends in Word Embedding Models

The word embedding models are perhaps some of the older and more mature models which have been developed starting with Word2Vec in 2013. The three most common models leveraging deep learning (unsupervised approaches) models based on embedding word vectors in a continuous vector space based on semantic and contextual similarity are:

- ***Word2Vec***
- ***GloVe***
- ***FastText***

These models are based on the principle of ***distributional hypothesis*** in the field of ***distributional semantics,*** which tells us that words which occur and are used in the same context, are semantically similar to one another and have similar meanings (*'a word is characterized by the company it keeps'*). Do refer to *my article on word embeddings* which cover these three methods in detail, if you are interested in the gory details!

Another interesting model in this area which has been developed recently, is ***ELMo***. This has been developed by the *Allen Institute for*

*Artificial Intelligence. ELMo* is a take on the famous muppet character of the same name from the famed show, 'Sesame Street', but actually is an acronym which stands for '***E****mbeddings from* ***L****anguage* ***Mo****dels*'.

Elmo from Sesame Street!

Basically, *ELMo* gives us word embeddings which are learnt from a deep bidirectional language model (biLM), which is typically pre-trained on a large text corpus, enabling transfer learning for these embeddings to be used across different NLP tasks. Allen AI tells us that ELMo representations are contextual, deep and character-based which uses morphological clues to form representations even for OOV (out-of-vocabulary) tokens.

## Trends in Universal Sentence Embedding Models

The concept of sentence embeddings is not a very new concept, because back when word embeddings were built, one of the easiest ways to build a baseline sentence embedding model was by averaging.

A ***baseline sentence embedding model*** can be built by just averaging out the individual word embeddings for every sentence\document (kind of similar to bag of words, where we lose that inherent context and sequence of words in the sentence). We do cover this in detail *in my article*. The following figure shows a way of implementing this.

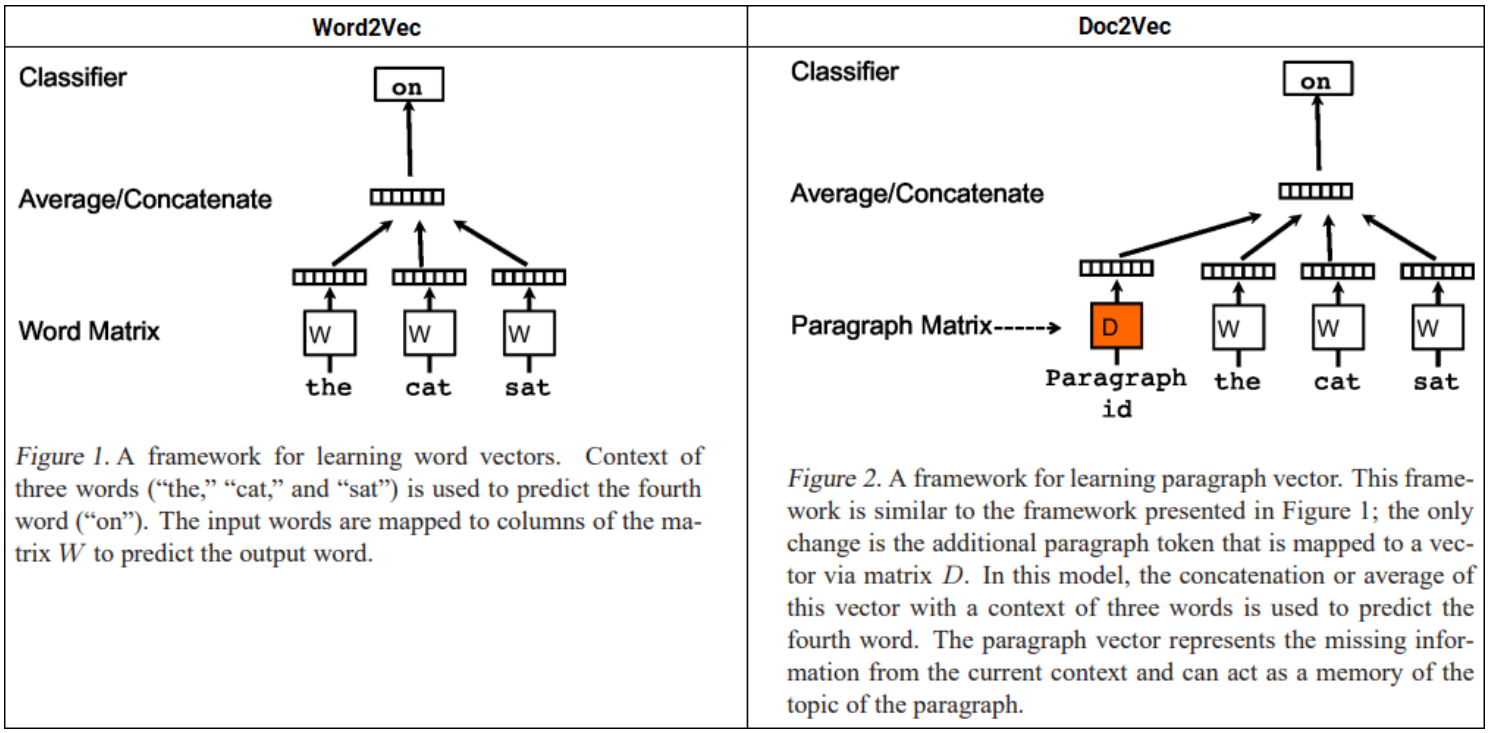
```python
def average_word_vectors(words, model, vocabulary, num_features):

    feature_vector = np.zeros((num_features,),dtype="float64")
    nwords = 0.

    for word in words:
        if word in vocabulary:
            nwords = nwords + 1.
            feature_vector = np.add(feature_vector, model[word])

    if nwords:
        feature_vector = np.divide(feature_vector, nwords)

    return feature_vector


def averaged_word_vectorizer(corpus, model, num_features):
    vocabulary = set(model.wv.index2word)
    features = [average_word_vectors(tokenized_sentence, model, vocabulary, num_features)
                for tokenized_sentence in corpus]
    return np.array(features)


# get document level embeddings
w2v_feature_array = averaged_word_vectorizer(corpus=tokenized_corpus, model=w2v_model,
                                             num_features=feature_size)
pd.DataFrame(w2v_feature_array)
```

feature_engg_text_31.py hosted with ❤ by GitHub view raw

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0.004690 | 0.009370 | -0.009667 | 0.026014 | 0.034989 | 0.010402 | -0.033441 | -0.011956 | -0.000243 | 0.010552 |
| 1 | 0.005751 | 0.003210 | -0.001964 | 0.016550 | 0.030962 | 0.004340 | -0.019463 | -0.009149 | 0.008256 | 0.019600 |
| 2 | 0.016712 | 0.004806 | -0.001924 | -0.027226 | 0.029162 | -0.017201 | -0.023197 | -0.008610 | -0.011976 | 0.020602 |
| 3 | -0.009216 | 0.003900 | -0.009232 | -0.005232 | 0.042718 | -0.032432 | -0.006243 | 0.013524 | 0.008095 | 0.021227 |
| 4 | -0.016321 | -0.008715 | -0.001633 | -0.000501 | 0.027367 | -0.037861 | 0.008515 | 0.021066 | 0.020373 | 0.016512 |
| 5 | 0.018538 | 0.007522 | -0.009302 | -0.025440 | 0.037199 | -0.009890 | -0.021419 | -0.011769 | -0.002221 | 0.018277 |
| 6 | 0.008532 | 0.008041 | -0.016573 | 0.018653 | 0.036140 | 0.004038 | -0.022891 | 0.000484 | -0.005900 | 0.015766 |
| 7 | 0.024419 | 0.012915 | -0.010596 | -0.039350 | 0.037018 | -0.013378 | -0.020677 | -0.004417 | -0.011864 | 0.013540 |

Of course, there are more sophisticated approaches like encoding sentences in a linear weighted combination of their word embeddings and then removing some of the common principal components. Do check out, *'A Simple but Tough-to-Beat Baseline for Sentence Embeddings'*.

***Doc2Vec*** is also a very popular approach proposed by Mikolov et. al. in their paper *'Distributed Representations of Sentences and Documents'*. Herein, they propose the Paragraph Vector, an unsupervised algorithm that learns fixed-length feature embeddings from variable-length pieces of texts, such as sentences, paragraphs, and documents.




Word2Vec vs. Doc2Vec (Source: https://arxiv.org/abs/1405.4053)

Based on the above depiction, the model represents each document by a dense vector which is trained to predict words in the document. The only difference being the paragraph or document ID, used along with the regular word tokens to build out the embeddings. Such a design enables this model to overcome the weaknesses of bag-of-words models.

***Neural-Net Language Models (NNLM)*** is a very early idea based on a neural probabilistic language model proposed by Bengio et al. in their paper, *'A Neural Probabilistic Language Model'* in 2003, they talk about learning a distributed representation for words which allows each training sentence to inform the model about an exponential number of semantically neighboring sentences. The model learns simultaneously a distributed representation for each word along with the probability function for word sequences, expressed in terms of these representations. Generalization is obtained because a sequence of words that has never been seen before gets high probability if it is made of words that are similar (in the sense of having a nearby representation) to words forming an already seen sentence.

Google has built a universal sentence embedding model, **nnlm-en-dim128** which is a token-based text embedding-trained model that uses a three-hidden-layer feed-forward Neural-Net Language Model on the English Google News 200B corpus. This model maps any body of text into 128-dimensional embeddings. *We will be using this in our hands-on demonstration shortly!*

***Skip-Thought Vectors*** were also one of the first models in the domain of unsupervised learning-based generic sentence encoders. In their proposed paper, *'Skip-Thought Vectors'*, using the continuity of text from books, they have trained an encoder-decoder model that tries to reconstruct the surrounding sentences of an encoded passage. Sentences that share semantic and syntactic properties are mapped to similar vector representations.

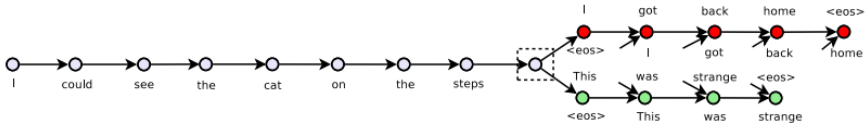
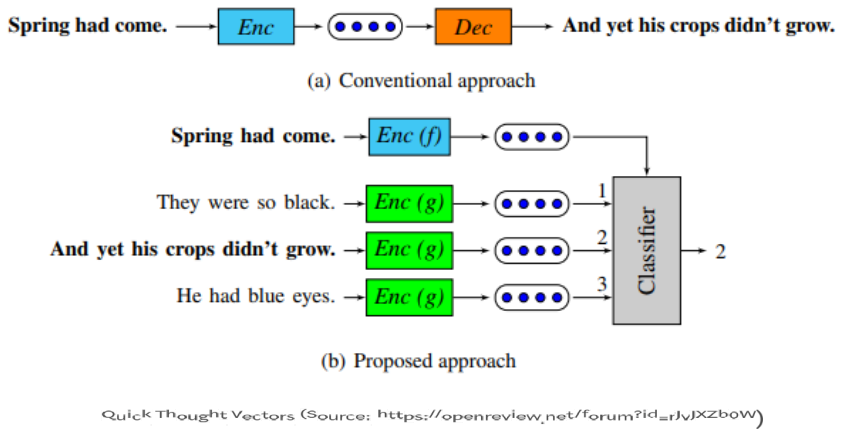



Figure 1: The skip-thoughts model. Given a tuple $(s_{i-1}, s_i, s_{i+1})$ of contiguous sentences, with $s_i$ the $i$-th sentence of a book, the sentence $s_i$ is encoded and tries to reconstruct the previous sentence $s_{i-1}$ and next sentence $s_{i+1}$. In this example, the input is the sentence triplet *I got back home. I could see the cat on the steps. This was strange.* Unattached arrows are connected to the encoder output. Colors indicate which components share parameters. ⟨eos⟩ is the end of sentence token.

Skip_Thought Vectors (Source: https://arxiv.org/abs/1506.06726)

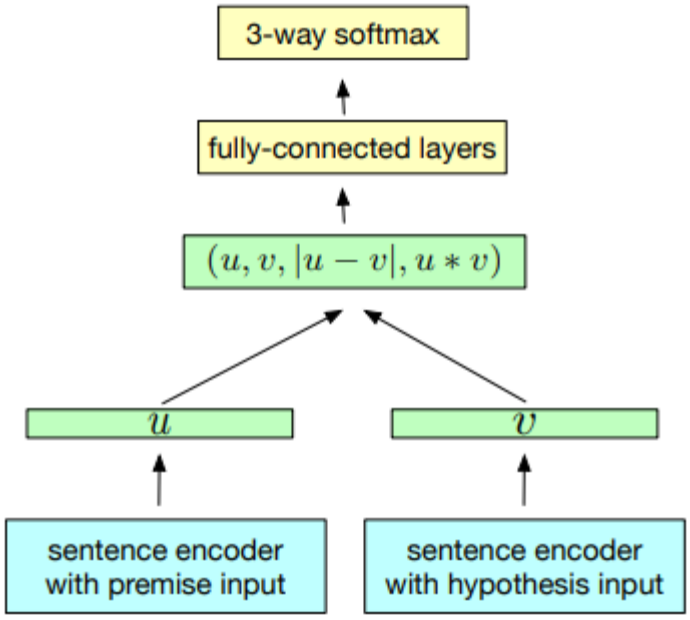
This is just like the Skip-gram model, but for sentences, where we try to predict the surrounding sentences of a given source sentence.

***Quick Thought Vectors*** is a more recent unupervised approach towards learning sentence emebddings. Details are mentioned in the paper *'An efficient framework for learning sentence representations'*. Interestingly, they reformulate the problem of predicting the context in which a sentence appears as a classification problem by replacing the decoder with a classfier in the regular encoder-decoder architecture.




Thus, given a sentence and the context in which it appears, a classifier distinguishes context sentences from other contrastive sentences based on their embedding representations. Given an input sentence, it is first encoded by using some function. But instead of generating the target sentence, the model chooses the correct target sentence from a set of candidate sentences. Viewing generation as choosing a sentence from all possible sentences, this can be seen as a discriminative approximation to the generation problem.

***InferSent*** is interestingly a supervised learning approach to learning universal sentence embeddings using natural language inference data. This is hardcore supervised transfer learning, where just like we get pre-trained models trained on the ImageNet dataset for computer vision, they have universal sentence representations trained using supervised data from the Stanford Natural Language Inference datasets. Details are mentioned in their paper, *'Supervised Learning of Universal Sentence Representations from Natural Language Inference Data'*. The dataset used by this model is the SNLI dataset that comprises 570k human-generated English sentence pairs, manually labeled with one of the three categories: entailment, contradiction and neutral. It captures natural language inference useful for understanding sentence semantics.




InferSent training scheme (Source: https://arxiv.org/abs/1705.02364)

Based on the architecture depicted in the above figure, we can see that it uses a shared sentence encoder that outputs a representation for the premise $\boldsymbol{u}$ and the hypothesis $\boldsymbol{v}$. Once the sentence vectors are generated, 3 matching methods are applied to extract relations between u and v :

- Concatenation $\boldsymbol{(u, v)}$
- Element-wise product $\boldsymbol{u * v}$
- Absolute element-wise difference $\boldsymbol{|u - v|}$

The resulting vector is then fed into a 3-class classifier consisting of multiple fully connected layers culminating in a softmax layer.

***Universal Sentence Encoder*** from Google is one of the latest and best universal sentence embedding models which was published in early 2018! The Universal Sentence Encoder encodes any body of text into 512-dimensional embeddings that can be used for a wide variety of NLP tasks including text classification, semantic similarity and clustering. It is trained on a variety of data sources and a variety of tasks with the aim of dynamically accommodating a wide variety of natural language understanding tasks which require modeling the meaning of sequences of words rather than just individual words.

Their key finding is that, transfer learning using sentence embeddings tends to outperform word embedding level transfer. Do check out their paper, *'Universal Sentence Encoder'* for further details. Essentially, they have two versions of their model available in *TF-Hub* as **universal-sentence-encoder**. Version 1 makes use of the transformer-network based sentence encoding model and Version 2 makes use of a Deep Averaging Network (DAN) where input embeddings for words and bi-grams are first averaged together and then passed through a feed-forward deep neural network (DNN) to produce sentence embeddings. We will be using Version 2 in our hands-on demonstration shortly.

## Understanding our Text Classification Problem

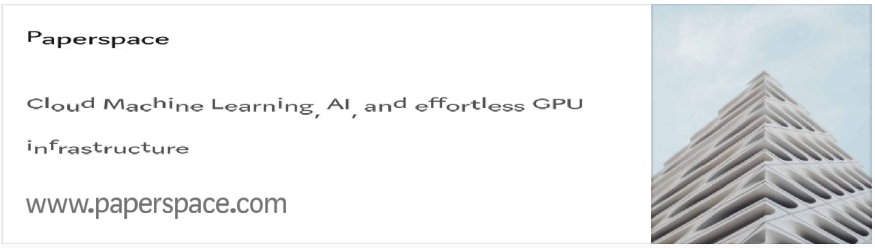
It's time for putting some of these universal sentence encoders into action with a hands-on demonstration! Like the article mentions, the premise of our demonstration today will focus on a very popular NLP task, text classification—in the context of sentiment analysis. We will be working with the benchmark ***IMDB Large Movie Review Dataset***. Feel free to download it *here or* you can even download it from ***my GitHub repository***.

This dataset comprises a total of 50,000 movie reviews, where 25K have positive sentiment and 25K have negative sentiment. We will be training our models on a total of 30,000 reviews as our training dataset, validate on 5,000 reviews and use 15,000 reviews as our test dataset. The main objective is to correctly predict the sentiment of each review as either positive or negative.

## Universal Sentence Embeddings in Action

Now that we have our main objective cleared up, let's put universal sentence encoders into action! The entire tutorial is available in ***my GitHub repository*** as a ***Jupyter Notebook***. Feel free to *download it* and play around with it. I recommend using a GPU-based instance for

playing around with this. I love using ***Paperspace*** where you can spin up notebooks in the cloud without needing to worry about configuring instances manually.




My setup was an 8 CPU, 30 GB, 250 GB SSD and an NVIDIA Quadro P4000 which is usually cheaper than most AWS GPU instances (I love AWS though!).

> ***Note:*** *This tutorial is built using TensorFlow entirely given that they provide an easy access to the sentence encoders. However I'm not a big fan of their old APIs and I'm looking for someone to assist me on re-implementing the code using the* `tf.keras` *APIs instead of* `tf.estimator` *. Do reach out to me if you are interested in contributing and we can even feature your work on the same! (contact links in my profile and in the footer)*

## Load up Dependencies

We start by installing `tensorflow-hub` which enables us to use these sentence encoders easily.

```
In [1]: !pip install tensorflow-hub
```

```
Collecting tensorflow-hub
  Downloading https://files.pythonhosted.org/packages/5f/22/64f246ef80e64b1a13b2f463cefa44f397a51c49a303294f5f3d04ac39ac/tens
    100% |################################| 61kB 8.5MB/s ta 0:00:011
Requirement already satisfied: numpy>=1.12.0 in /usr/local/lib/python3.6/dist-packages (from tensorflow-hub) (1.14.3)
Requirement already satisfied: six>=1.10.0 in /usr/local/lib/python3.6/dist-packages (from tensorflow-hub) (1.11.0)
Requirement already satisfied: protobuf>=3.4.0 in /usr/local/lib/python3.6/dist-packages (from tensorflow-hub) (3.5.2.post1)
Requirement already satisfied: setuptools in /usr/local/lib/python3.6/dist-packages (from protobuf>=3.4.0->tensorflow-hub) (
Installing collected packages: tensorflow-hub
Successfully installed tensorflow-hub-0.1.1
```

Let's now load up our essential dependencies for this tutorial!

```
import tensorflow as tf
import tensorflow_hub as hub
import numpy as np
import pandas as pd
```

The following commands help you check if `tensorflow` will be using a GPU (if you have one set up already!)

```
In [12]: tf.test.is_gpu_available()
Out[12]: True

In [13]: tf.test.gpu_device_name()
Out[13]: '/device:GPU:0'
```

## Load and View Dataset

We can now load up out dataset and view it using `pandas`. I provide a compressed version of the dataset in my repository which you can use as follows.

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 50000 entries, 0 to 49999
Data columns (total 2 columns):
review       50000 non-null object
sentiment    50000 non-null object
dtypes: object(2)
memory usage: 781.3+ KB
```

We encode the sentiment column as 1s and 0s just to make things easier for us during model development (label encoding).

| | review | sentiment |
|---|---|---|
| 0 | One of the other reviewers has mentioned that ... | 1 |
| 1 | A wonderful little production. <br /><br />The... | 1 |
| 2 | I thought this was a wonderful way to spend ti... | 1 |
| 3 | Basically there's a family where a little boy ... | 0 |
| 4 | Petter Mattei's "Love in the Time of Money" is... | 1 |

Our movie review dataset

### Building Train, Validation and Test Datasets

We will now create train, validation and test datasets before we start modeling. We will use 30,000 reviews for train, 5,000 for validation and 15,000 for test. You can use a train-test splitting function also like `train_test_split()` from `scikit-learn`. I was just lazy and subsetted the dataset using simple list slicing.

```
((30000,), (5000,), (15000,))
```

## Basic Text Wrangling

There is some basic text wrangling and pre-processing we need to do to remove some noise from our text like contractions, unnecessary special characters, HTML tags and so on. The following code helps us build a simple, yet effective text wrangling system. Do install the following libraries in case you don't have them.

```
In [17]: !pip install contractions
         !pip install beautifulsoup4

         Requirement already satisfied: contractions in /usr/local/lib/python3.6/dist-packages (0.0.17)
         You are using pip version 10.0.1, however version 18.1 is available.
         You should consider upgrading via the 'pip install --upgrade pip' command.
         Requirement already satisfied: beautifulsoup4 in /usr/local/lib/python3.6/dist-packages (4.6.3)
         You are using pip version 10.0.1, however version 18.1 is available.
         You should consider upgrading via the 'pip install --upgrade pip' command.
```

The following functions help us build our text wrangling system.

Let's now pre-process our datasets using the function we implemented above.

### Build Data Ingestion Functions

Since we will be implementing our models in `tensorflow` using the `tf.estimator` API, we need to define some functions to build data and feature engineering pipelines to enable data flowing into our models during training. The following functions will help us. We leverage the `numpy_input_fn()` which helps in feeding a dict of numpy arrays into the model.

We are now ready to build our models!

### Build Deep Learning Model with Universal Sentence Encoder

We need to first define the sentence embedding feature which leverages the universal sentence encoder before building the model. We can do that using the following code.

```
INFO:tensorflow:Using /tmp/tfhub_modules to cache modules.
```

Like we discussed, we use the Universal Sentence Encoder Version 2 and it works on the `sentence` attribute in our input dictionary which will be a `numpy` array of our reviews. We will build a simple feed-forward DNN now with two hidden layers. Just a standard model, nothing too sophisticated since we want to see how well these embeddings perform even on a simple model. Here we are leveraging transfer learning in the form of pre-trained embeddings. We are not fine-tuning here by keeping the embedding weights fixed by setting `trainable=False`.

We had set our `batch_size` to **256** and we will be flowing in data in batches of **256** records for **1500 steps** which translates to roughly **12–13 epochs**.

## Model Training

Let's train our model now on our training dataset and evaluate on both train and validation datasets at steps of 100.

```
------------------------------------------------------------
--------
Training for step = 0
Train Time (s): 78.62789511680603
Eval Metrics (Train): {'accuracy': 0.84863335,
'accuracy_baseline': 0.5005, 'auc': 0.9279859,
'auc_precision_recall': 0.92819566, 'average_loss':
0.34581015, 'label/mean': 0.5005, 'loss': 44.145977,
'precision': 0.86890674, 'prediction/mean': 0.47957155,
'recall': 0.8215118, 'global_step': 100}
Eval Metrics (Validation): {'accuracy': 0.8454,
'accuracy_baseline': 0.505, 'auc': 0.92413086,
'auc_precision_recall': 0.9200026, 'average_loss':
0.35258815, 'label/mean': 0.495, 'loss': 44.073517,
'precision': 0.8522351, 'prediction/mean': 0.48447067,
'recall': 0.8319192, 'global_step': 100}

------------------------------------------------------------
--------
Training for step = 100
Train Time (s): 76.1651611328125
Eval Metrics (Train): {'accuracy': 0.85436666,
'accuracy_baseline': 0.5005, 'auc': 0.9321357,
'auc_precision_recall': 0.93224275, 'average_loss':
0.3330773, 'label/mean': 0.5005, 'loss': 42.520508,
'precision': 0.8501513, 'prediction/mean': 0.5098621,
'recall': 0.86073923, 'global_step': 200}
Eval Metrics (Validation): {'accuracy': 0.8494,
'accuracy_baseline': 0.505, 'auc': 0.92772096,
'auc_precision_recall': 0.92323804, 'average_loss':
0.34418356, 'label/mean': 0.495, 'loss': 43.022945,
'precision': 0.83501947, 'prediction/mean': 0.5149463,
'recall': 0.86707073, 'global_step': 200}

------------------------------------------------------------
--------
...
...
...
------------------------------------------------------------
--------
Training for step = 1400
Train Time (s): 85.99037742614746
Eval Metrics (Train): {'accuracy': 0.8783,
'accuracy_baseline': 0.5005, 'auc': 0.9500882,
'auc_precision_recall': 0.94986326, 'average_loss':
```

```
0.28882334, 'label/mean': 0.5005, 'loss': 36.871063,
'precision': 0.865308, 'prediction/mean': 0.5196238,
'recall': 0.8963703, 'global_step': 1500}
Eval Metrics (Validation): {'accuracy': 0.8626,
'accuracy_baseline': 0.505, 'auc': 0.93708724,
'auc_precision_recall': 0.9336051, 'average_loss':
0.32389137, 'label/mean': 0.495, 'loss': 40.486423,
'precision': 0.84044176, 'prediction/mean': 0.5226699,
'recall': 0.8917172, 'global_step': 1500}

----------------------------------------------------------
--------
Training for step = 1500
Train Time (s): 86.91469407081604
Eval Metrics (Train): {'accuracy': 0.8802,
'accuracy_baseline': 0.5005, 'auc': 0.95115364,
'auc_precision_recall': 0.950775, 'average_loss': 0.2844779,
'label/mean': 0.5005, 'loss': 36.316326, 'precision':
0.8735527, 'prediction/mean': 0.51057553, 'recall':
0.8893773, 'global_step': 1600}
Eval Metrics (Validation): {'accuracy': 0.8626,
'accuracy_baseline': 0.505, 'auc': 0.9373224,
'auc_precision_recall': 0.9336302, 'average_loss':
0.32108024, 'label/mean': 0.495, 'loss': 40.135033,
'precision': 0.8478599, 'prediction/mean': 0.5134171,
'recall': 0.88040406, 'global_step': 1600}
```

I have highlighted the metrics of interest in the output logs above and as you can see, we get an overall **accuracy** of close to **87%** on our validation dataset and an **AUC** of **94%** which is quite good on such a simple model!

## Model Evaluation

Let's now evaluate our model and check the overall performance on the train and test datasets.

```
In [19]: dnn.evaluate(input_fn=predict_train_input_fn)
```

```
Out[19]: {'accuracy': 0.8802,
          'accuracy_baseline': 0.5005,
          'auc': 0.95115364,
          'auc_precision_recall': 0.950775,
          'average_loss': 0.2844779,
          'label/mean': 0.5005,
          'loss': 36.316326,
          'precision': 0.8735527,
          'prediction/mean': 0.51057553,
          'recall': 0.8893773,
          'global_step': 1600}
```

```
In [20]: dnn.evaluate(input_fn=predict_test_input_fn)
```

```
Out[20]: {'accuracy': 0.8663333,
          'accuracy_baseline': 0.5006667,
          'auc': 0.9406502,
          'auc_precision_recall': 0.93988097,
          'average_loss': 0.31214723,
          'label/mean': 0.5006667,
          'loss': 39.679733,
          'precision': 0.8597569,
          'prediction/mean': 0.5120608,
          'recall': 0.8758988,
          'global_step': 1600}
```

We get an overall accuracy of close to 87% on the test data giving us consistent results based on what we observed on our validation dataset earlier! Thus, this should give you an idea of how easy it is to leverage pre-trained universal sentence embeddings and not worry about the hassle of feature engineering or complex modeling.

## Bonus: Transfer Learning with Different Universal Sentence Embeddings

Let's now try building different deep learning classifiers based on different sentence embeddings. We will try the following:

- NNLM-128
- USE-512

We will also cover the two most prominent methodologies for transfer learning here.

- Build a model using freezed pre-trained sentence embeddings
- Build a model where we fine-tune and update the pre-trained sentence embeddings during training

The following generic function can plug and play different universal sentence encoders from `tensorflow-hub` !

We can now train our models using the above-defined approaches.

```
============================================================
========
Training with https://tfhub.dev/google/nnlm-en-dim128/1
Trainable is: False
============================================================
========
------------------------------------------------------------
--------
Training for step = 0
Train Time (s): 30.525171756744385
Eval Metrics (Train): {'accuracy': 0.8480667, 'auc':
0.9287864, 'precision': 0.8288572, 'recall': 0.8776557}
Eval Metrics (Validation): {'accuracy': 0.8288, 'auc':
0.91452694, 'precision': 0.7999259, 'recall': 0.8723232}
------------------------------------------------------------
--------
...
...
------------------------------------------------------------
--------
Training for step = 1500
Train Time (s): 28.242169618606567
Eval Metrics (Train): {'accuracy': 0.8616, 'auc': 0.9385461,
'precision': 0.8443543, 'recall': 0.8869797}
```

```
Eval Metrics (Validation): {'accuracy': 0.828, 'auc':
0.91572505, 'precision': 0.80322945, 'recall': 0.86424243}


============================================================
========
Training with https://tfhub.dev/google/nnlm-en-dim128/1
Trainable is: True
============================================================
========
------------------------------------------------------------
--------
Training for step = 0
Train Time (s): 45.97756814956665
Eval Metrics (Train): {'accuracy': 0.9997, 'auc': 0.9998141,
'precision': 0.99980015, 'recall': 0.9996004}
Eval Metrics (Validation): {'accuracy': 0.877, 'auc':
0.9225529, 'precision': 0.86671925, 'recall': 0.88808084}
------------------------------------------------------------
--------
...
...
------------------------------------------------------------
--------
Training for step = 1500
Train Time (s): 44.654765605926514
Eval Metrics (Train): {'accuracy': 1.0, 'auc': 1.0,
'precision': 1.0, 'recall': 1.0}
Eval Metrics (Validation): {'accuracy': 0.875, 'auc':
0.91479605, 'precision': 0.8661916, 'recall': 0.8840404}


============================================================
========
Training with https://tfhub.dev/google/universal-sentence-
encoder/2
Trainable is: False
============================================================
========
------------------------------------------------------------
--------
Training for step = 0
Train Time (s): 261.7671597003937
Eval Metrics (Train): {'accuracy': 0.8591, 'auc': 0.9373971,
'precision': 0.8820655, 'recall': 0.8293706}
Eval Metrics (Validation): {'accuracy': 0.8522, 'auc':
0.93081224, 'precision': 0.8631799, 'recall': 0.8335354}
------------------------------------------------------------
--------
...
...
------------------------------------------------------------
--------
Training for step = 1500
Train Time (s): 258.4421606063843
Eval Metrics (Train): {'accuracy': 0.88733333, 'auc':
0.9558296, 'precision': 0.8979955, 'recall': 0.8741925}
Eval Metrics (Validation): {'accuracy': 0.864, 'auc':
0.938815, 'precision': 0.864393, 'recall': 0.860202}
```

```
================================================================
========
Training with https://tfhub.dev/google/universal-sentence-
encoder/2
Trainable is: True
================================================================
========
----------------------------------------------------------------
--------
Training for step = 0
Train Time (s): 313.1993100643158
Eval Metrics (Train): {'accuracy': 0.99916667, 'auc':
0.9996535, 'precision': 0.9989349, 'recall': 0.9994006}
Eval Metrics (Validation): {'accuracy': 0.9056, 'auc':
0.95068294, 'precision': 0.9020474, 'recall': 0.9078788}
----------------------------------------------------------------
--------
...
...
----------------------------------------------------------------
--------
Training for step = 1500
Train Time (s): 305.9913341999054
Eval Metrics (Train): {'accuracy': 1.0, 'auc': 1.0,
'precision': 1.0, 'recall': 1.0}
Eval Metrics (Validation): {'accuracy': 0.9032, 'auc':
0.929281, 'precision': 0.8986784, 'recall': 0.9066667}
```

I've depicted the evaluation metrics of importance in the above outputs, and you can see we definitely get some good results with our models. The following table summarizes these comparative results in a nice way.

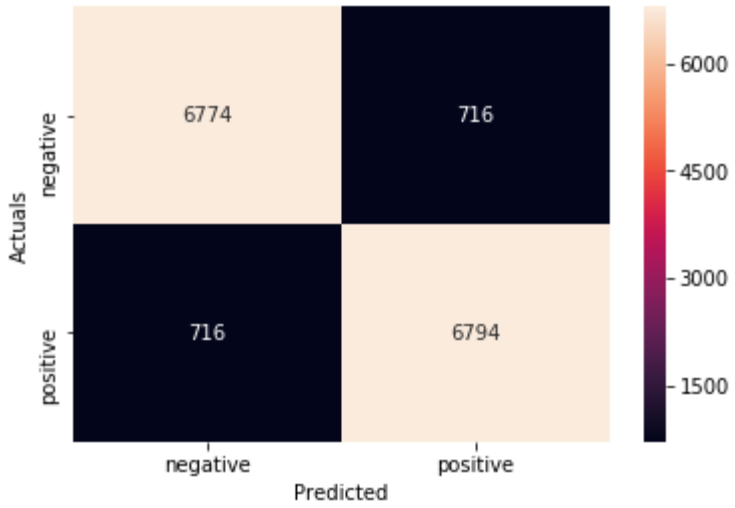
| | Model Dir | Training Accuracy | Test Accuracy | Training AUC | Test AUC | Training Precision | Test Precision | Training Recall | Test Recall |
|---|---|---|---|---|---|---|---|---|---|
| **nnlm-en-dim128** | /storage/models/nnlm-en-dim128_f/ | 0.861600 | 0.836133 | 0.938546 | 0.918221 | 0.844354 | 0.822770 | 0.886980 | 0.857390 |
| **nnlm-en-dim128-with-training** | /storage/models/nnlm-en-dim128_t/ | 1.000000 | 0.878467 | 1.000000 | 0.919655 | 1.000000 | 0.875975 | 1.000000 | 0.882157 |
| **use-512** | /storage/models/use-512_f/ | 0.887333 | 0.867067 | 0.955830 | 0.942319 | 0.897995 | 0.876776 | 0.874192 | 0.854594 |
| **use-512-with-training** | /storage/models/use-512_t/ | 1.000000 | 0.904533 | 1.000000 | 0.930401 | 1.000000 | 0.904660 | 1.000000 | 0.904660 |

Comparing results from different Universal Sentence Encoders

Looks like Google's Universal Sentence Encoder with fine-tuning gave us the best results on the test data. Let's load up this saved model and run an evaluation on the test data.

```
[0, 1, 0, 1, 1, 0, 1, 1, 1, 1]
```

One of the best ways to evaluate our model performance is to visualize the model predictions in the form of a confusion matrix.




Confusion Matrix from our Best Model Predictions

We can also print out the model's classification report using scikit-learn to show the other important metrics which can be derived from the confusion matrix including precision, recall and f1-score.

| | precision | recall | f1-score | support |
|---|---|---|---|---|
| negative | 0.90 | 0.90 | 0.90 | 7490 |
| positive | 0.90 | 0.90 | 0.90 | 7510 |
| avg / total | 0.90 | 0.90 | 0.90 | 15000 |

Model Performance Metrics on Test Data

We obtain an overall model **accuracy** and **f1-score** of **90%** on the test data which is really good! Go ahead and try this out and maybe get an even better score and let me know about it!

## Conclusion and Future Scope

Universal Sentence Embeddings are definitely a huge step forward in enabling transfer learning for diverse NLP tasks. In fact, we have seen models like ELMo, Universal Sentence Encoder, ULMFiT have indeed made headlines by showcasing that pre-trained models can be used to achieve state-of-the-art results on NLP tasks. Famed Research Scientist and Blogger Sebastian Ruder, mentioned the same in his recent tweet based on *a very interesting article* which he wrote recently.

> New piece about a direction I'm super excited about: NLP's ImageNet moment has arrived @gradientpub https://t.co/Kb3szva7tC
>
> —@seb_ruder

I'm definitely excited about what the future holds for generalizing NLP even further, and enabling us to solve complex tasks with ease!

. . .

The code used for hands-on demonstrations in this article is available in ***my GitHub repository*** as a ***Jupyter Notebook*** which you can play around with!

**Help Needed:** Like I mentioned, I'm looking for someone to help me convert this code to use the newer `tf.keras` APIs instead of `tf.estimator`. Interested? Reach out to me!

Have feedback for me? Or interested in working with me on research, data science, artificial intelligence or even publishing an article on ***TDS***? You can reach out to me on **LinkedIn.**

Dipanjan Sarkar - AI Consultant & Data Science Mentor - Springboard | LinkedIn

View Dipanjan Sarkar's profile on LinkedIn, the world's largest professional community. Dipanja...

www.linkedin.co

Thanks to *Durba* for editing this article.